Classes and Dynamic Memory

THURSDAY, MAY 8

Section materials curated by Jonathan Coronado, Yasmine Alonso, and Sean Szumlanski, drawing upon materials from previous quarters.

This week's section exercise consists of two larger problems that will give you practice with designing classes and working with dynamic array allocation. These problems will help you get practice with the skills that you need for the next assignment, where you will start to implement your very own data structures! As you work on these problems, you may find this Classes and Objects Syntax Sheet to be helpful.

Remember that every week we will also be releasing a Qt Creator project containing starter code and testing infrastructure for that week's section problems. When a problem name is followed by the name of a **.cpp** file, that means you can practice writing the code for that problem in the named file of the Qt Creator project. Here is the zip of the section starter code:

📦 Starter project

1. Some Pointers on Cats

Topics: Pointer tracing and memory diagrams

Trace through the following function and draw the program's memory at the designated spot. Indicate which variables are on the stack and which are on the heap, and indicate orphaned memory. Indicate with a question mark (?) memory that we don't know the values of.

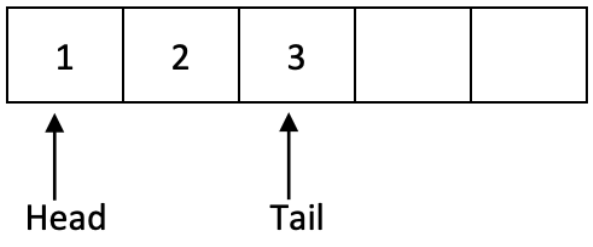
```
struct Lion {
    int roar;
    int *meow;
};

struct Savanna {
    int giraffe;
    Lion cat;
};

void explore(Savanna *prairie) {
    Lion *leader = &(prairie->cat);
    leader->meow = new int(2);
    leader->roar = 2;
    leader->meow = new int(3);
}

void kittens() {
    Savanna *habitat = new Savanna[3];
    habitat[1].giraffe = 3;
    explore(habitat);
    habitat[0].giraffe = 8;
    // DRAW THE MEMORY AS IT LOOKS HERE
}
```

2. The Notorious RBQ (**RingBufferQueue.h/.cpp**)

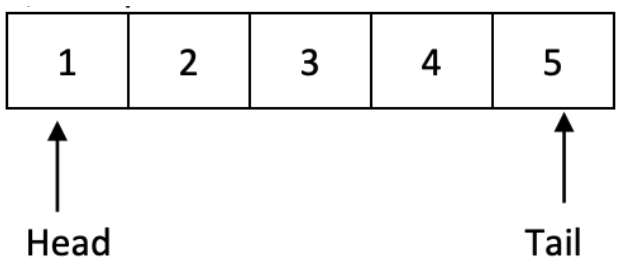
Topics: Classes, dynamic arrays

Think back to week 2 when we studied collections. We learned about Queues, a "first-in, first-

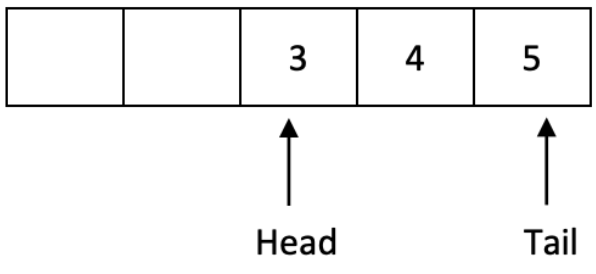
...ll data structure. Today in section, we're going to implement a special type of queue called a Ring Buffer Queue. A Ring Buffer Queue, or RBQ, is implemented by using an underlying array. ...ffer Queue, the capacity is capped; once the array is full, additional elements cannot be added until something is dequeued. Another "interesting" thing about RBQs is that we don't want to shift elements when an element is enqueued or dequeued. Instead, we want to keep track of the front and tail of the Queue. For example, say our queue can hold 5 elements and we enqueue 3 elements: 1, 2, 3. Our queue would look like this:



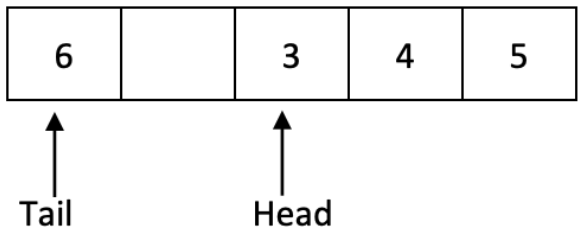
If we enqueued two more elements, our queue would then be full:



At this point, we cannot add any additional elements until we dequeue at least one element. Dequeuing will remove the element at head, and head will move onto the next element. If we dequeue 2 elements, our queue will look like this:



Now there's room to add more elements! Since we still don't want to shift any elements, adding an additional element will wrap around. So, if we enqueue an element, our queue will look like this:



Notice that the tail's index is less than the head's index!

Your job is to implement a **`RingBufferQueue`** class. Your class should have the following public methods:

Method	**Description**
`void enqueue(int elem)`	Enqueues **`elem`** if the queue has room; throws an error if queue is full
`int dequeue()`	Returns and removes the element at the front of the queue; throws a string exception if queue is empty
`int peek()`	Returns element at the front of the queue; throws a string exception if queue is empty
`bool isEmpty()`	Returns **`true`** if queue is empty and **`false`** otherwise
`bool isFull()`	Returns **`true`** if queue is full and **`false`** otherwise
`int size()`	Returns number of elements in the queue

You are welcome to add any private methods or fields that are necessary.

It can be hard to know where to start when writing an entire class, so we've given you this breakdown:

1. Start by identifying the private fields you will need, then write the constructor and `~RingBufferQueue()` to initialize the fields and do any cleanup, if necessary. Questions to think about:
    - Is it easier to keep track of head and tail (as pictured in the diagrams above)? Or would it be better to track head and size?
2. Write **`isEmpty()`**, **`isFull()`**, **`size()`**, and **`peek()`**. Questions to think about:
    - Which of these methods can be const? In general, how do you know when a method can be const?
3. Write **`enqueue()`** and **`dequeue()`**. Remember to handle error conditions! Questions to think about:
    - Can you call the methods from part 2 to reduce redundancy?
    - Would using modular math help with wrapping around?
    - Should either of these methods be const?
4. Finally, deal with ostream insertion!

If you want more practice with writing classes, think about how you could modify this class to implement a double-ended queue. (A double-ended queue, or deque, is one where you can enqueue and dequeue from either the front or the back).

---

3. Growing a Ring Buffer Queue (**`RingBufferQueue.h/.cpp`**)

Remember our good friend the `RingBufferQueue from earlier? Check out the problem definition from last week if you want a quick refresher. Last time we visited the RBQ it had fixed capacity – that is, it couldn't grow in size after it was initially created. How limiting! With our newfound pointer and dynamic allocation skills, we can remove this limitation on the RBQ and make it fully functional!

Add functionality to the **`RingBufferQueue`** from the previous section problem so that the queue resizes to an array twice as large when it runs out of space. In other words, if asked to enqueue when the queue is full, it will enlarge the array to give it enough capacity.

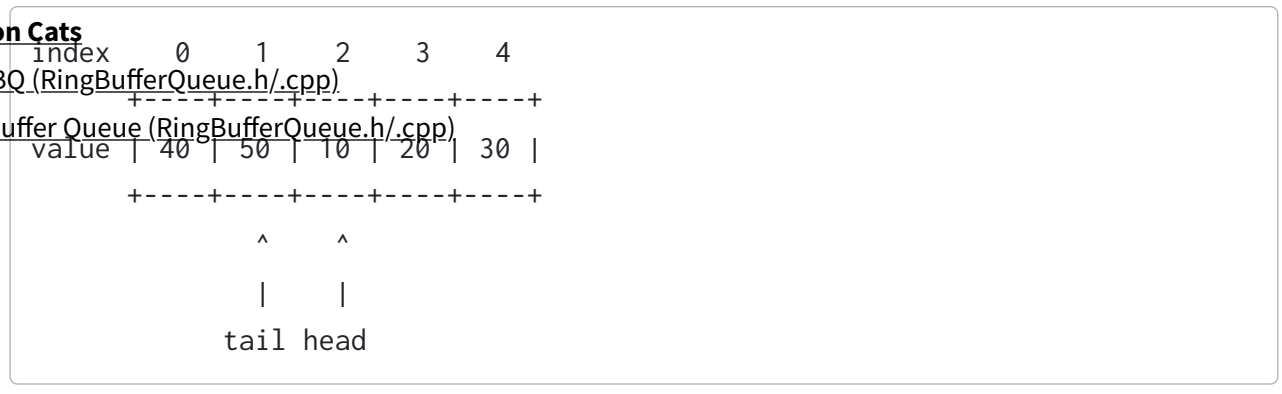
For example, say our queue can hold 5 elements and we enqueue the five values 10, 20, 30, 40, and 50. Our queue would look like this:

```
index    0    1    2    3    4
      +----+----+----+----+----+
value | 10 | 20 | 30 | 40 | 50 |
      +----+----+----+----+----+
         ^                   ^
         |                   |
       head                tail
```

If the client tries to enqueue a sixth element of 60, your improved queue class should grow to an array twice as large:

```
index    0    1    2    3    4    5    6    7    8    9
      +----+----+----+----+----+----+----+----+----+----+
value | 10 | 20 | 30 | 40 | 50 | 60 |    |    |    |    |
      +----+----+----+----+----+----+----+----+----+----+
         ^                        ^
         |                        |
       head                     tail
```

The preceding is the simpler case to handle. But what about if the queue has wrapped around via a series of enqueues and dequeues? For example, if the queue stores the following five elements:



If the client tries to add a sixth element of 60, you cannot simply copy the array as it is shown. If you do so, the head and wrapping will be broken. Instead, copy into the new array so that index 0 always becomes the new head of the queue. The picture will look the same as the previous one with the value 60 at index 5.

Write up the implementation of the new and improved **`RingBufferQueue`**. It should have the same members as in the previous problem, but with the new resizing behavior added. You may add new member functions to your class, but you should make them private.

Website programming by Julie Zelenski with modifications by Sean Szumlanski • Styles adapted from Chris Piech • This page last updated 2025-May-05